Solving Random Resistor Networks

From: *Aleksandar Donev*

To: ***Phillip Duxbury, Bruce Hendrickson, Sivan Toledo, Francois Pellegrini, and whoever else cares.***

These are some of my observations and directions for future work related to solving the random-resistor network problem in physics, which is at the core of an algorithm for doing non-linear network optimization included in a library that I am writing (SSCNO). I hope to get some comments and help from you, as well as contribute to your own work on preconditioning of sparse systems.

Physical Problem

I was studying iterative solutions to linear systems arising in network optimization:

$$(AC_AA^T)x = b \qquad \#$$

where A is the node-arc incidence matrix of a graph G, and C_A is a diagonal matrix containing the conductances (inverse resistances) of the arcs in G, x is a vector of (excess) potentials at the nodes and b is a vector of (excess) flows in/out of the nodes in G. The system matrix $C = AC_AA^T$ is the *conductance matrix* of the network and it has the sparcity structure of the Laplacian of G. The number of nodes of G is n, and the number of arcs is $m = O(n)$.

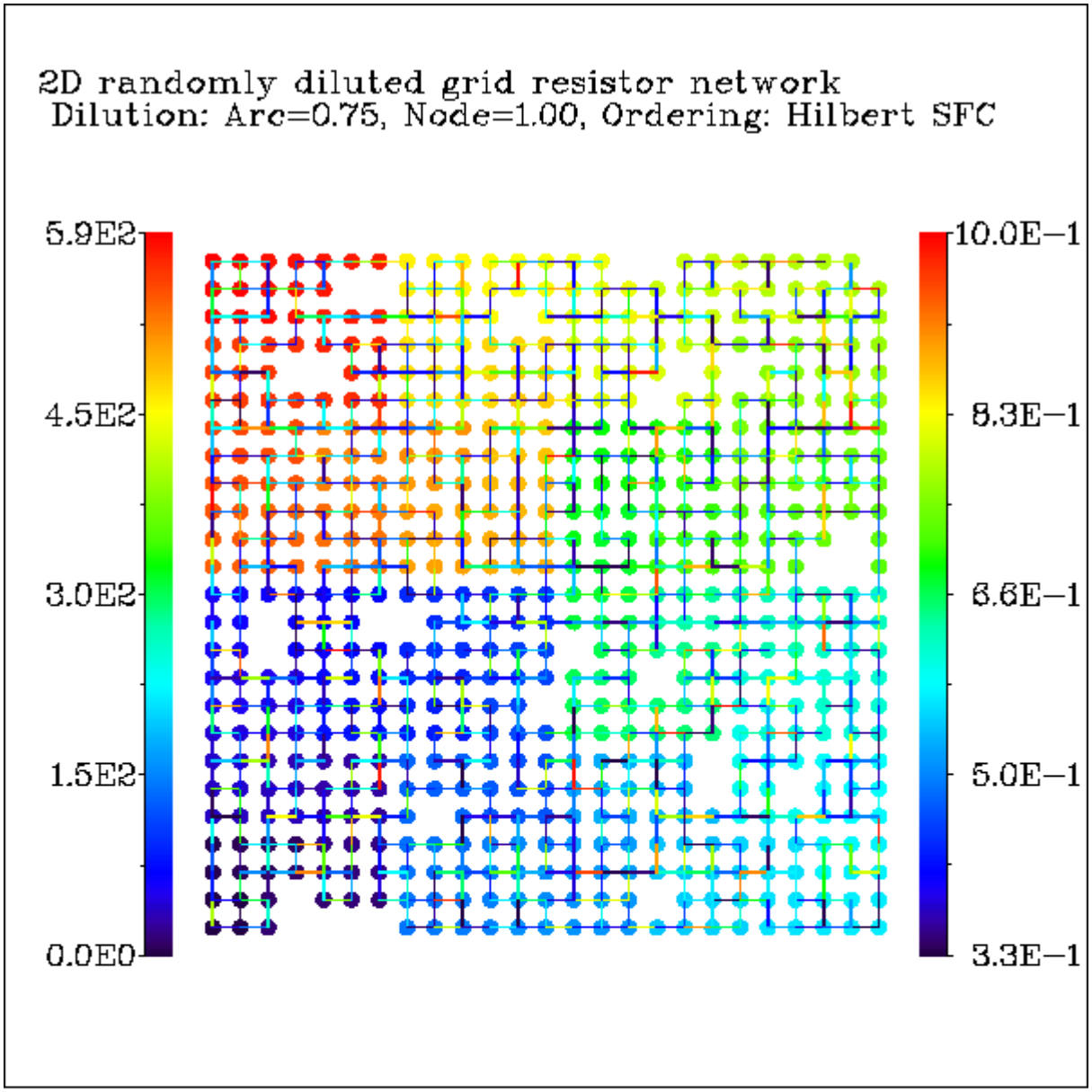
Our graphs are *randomly diluted hypercube lattices (grids)*. The *dilution d* tells how many arcs were randomly removed from a complete d-dimensional grid when G was created (a connected-backbone extraction is performed afterwards). So $d = 0.75$ means that 75% of the arcs were kept, and 25% of the arcs randomly chosen and diluted. The graph therefore becomes sparses and less like regular finite-difference grids as the dilution is lowered. In 2D the lower bound to get a connected graph is $d = 0.5$, and near this percolation point the graphs become fractal with some fascinating properties.

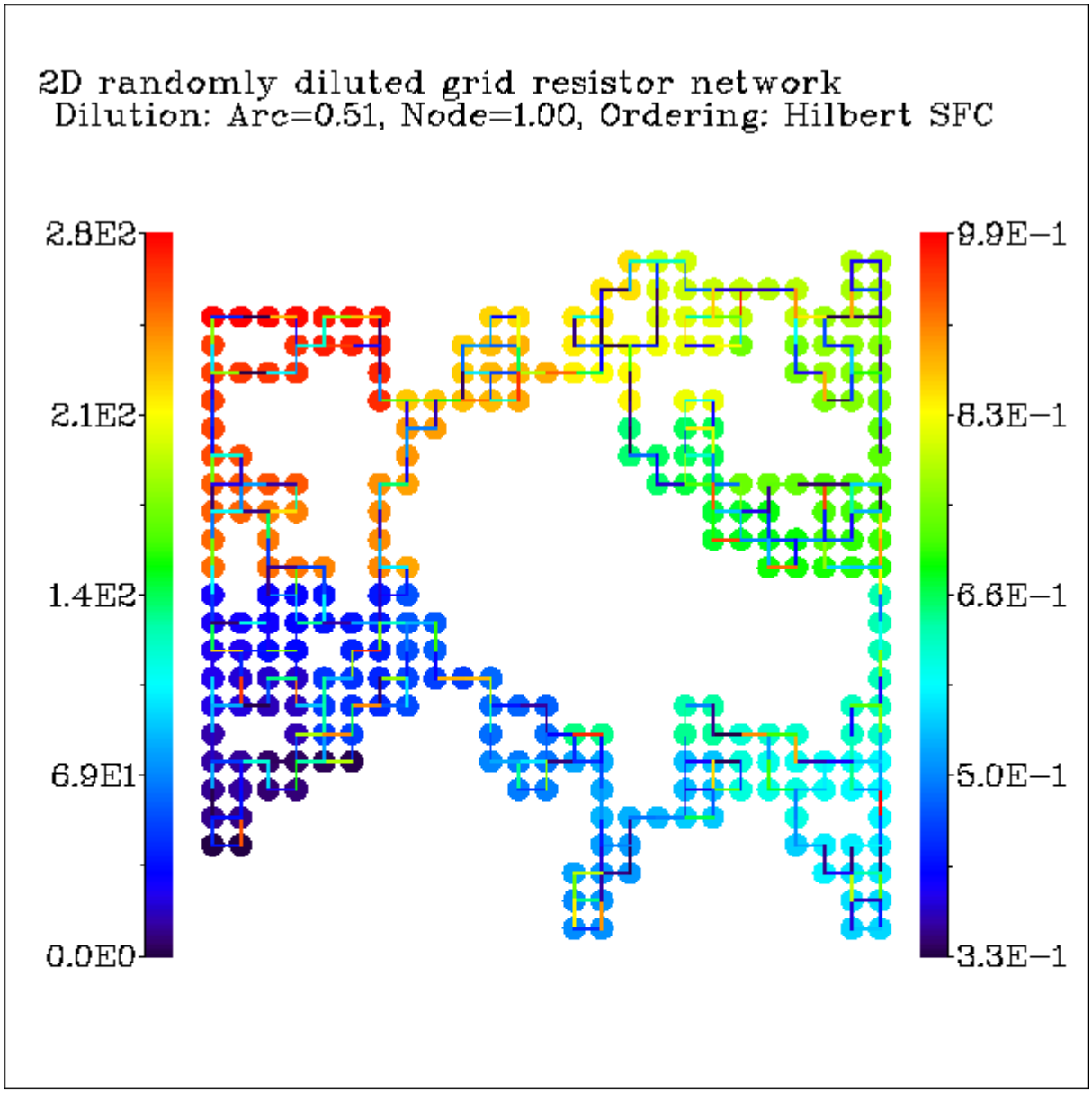
The conductances in C_A are randomly chosen, in these tests uniformly from some interval $[1, \kappa]$, where κ is a measure of the ill-conditioning in C_A. In certain physical applications, such as studying binary superconductor-normal metal mixtures, the range of conductances is very wide, some being very large, others very small.

Ill-conditioning of the conductance matrix C also comes because the unweighted Laplacian of G, $L = AA^T$, has a conditioning number which increases with the size of the grids. Physically, we expect that the number of CG iterations needed to converge to a good solutio of will be of the order $O(L)$, where L is the length of the physical system (so $n = L^{\dim}$, where dim is the dimensionality of the space). This is indeed observed. Near percolation, however, the paths between nodes are fractal and a one observes “critical slowdown” in conjugate-gradient codes.We can also expect significant differences between 2 and 3 dimensional graphs, as in FEM applications. Most of my experiments were done in 2D!

Pictures of two of my 2D graphs to help you in visualization, where the conductances of the arcs are shown as their color, for $d = 0.505$ (close to percolation) and $d = .75$, are shown below. The color of the nodes indicates their numbering in the numbering of the nodes of the grid. This numbering is very important for cache-performance of the codes. In particular, we have decided not

to use any specially-optimized data structures to represent G, but rather use a standard edge-based representation: an array of *head and tail nodes* for each arc, of size $[2, m]$. This makes the matrix-vector product calculation Cx a very bad performer due to large memory traffic, and reordering the nodes helps, but not by much on my Pentium machine (a factor of 2-4 or so). I got the best results for node orderings based on space-filling Hilbert curves, which come naturally for regular grid graphs. I tried some orderings based on heaviest edge matching, which take much longer to construct, but did not do any good for these grid graphs.


2D randomly diluted grid resistor network
Dilution: Arc=0.75, Node=1.00, Ordering: Hilbert SFC
5.9E2
4.5E2
3.0E2
1.5E2
0.0E0
10.0E−1
8.3E−1
6.6E−1
5.0E−1
3.3E−1


2D randomly diluted grid resistor network
Dilution: Arc=0.51, Node=1.00, Ordering: Hilbert SFC
2.8E2
2.1E2
1.4E2
6.9E1
0.0E0
9.9E−1
8.3E−1
6.6E−1
5.0E−1
3.3E−1

Let me make two more important points about the kind of systems that we wish to study. The structure of the Laplacian's and grids is relatively similar to finite-element and finite-difference methods, so that we can use a lot of techniques developed in those two fields. But there is an important difference: In finite elements, the mesh is a fictional construct–one can always make a coarser or finer mesh and still get a physical solution. In our networks, *the disordered (random) discrete structure is what makes the physics of the problem*, so making finer or coarser grids is to some extent non-physical. Therefore certain multigrid or multilevel preconditioning techniques which work wonders for elliptic PDE's are not really of much use to us!

Another note to make is that I have software that can generate other kinds of lattices (for example ones that include diagonal links in the grid or second-nearest neighbour lattices, etc.), but I have not used these yet. Also, we can add periodic boundary conditions in any given coordinate direction, which makes the lattice a torus. This usually introduces more fill-in in direct factorizations then when using free boundary conditions, but not too much (at most 25% or so in 3D).

Special Requirements for the Linear Solver

There are two things specific to using inside a non-linear network optimization which are very important in this context. The first is that we need to solve the system *repeatedly* (say 10-1000 times) with changing C_A and with increasingly higher precision. In the beginning the precision is very low, so only a few CG iterations are needed to achieve the desired accuracy, but later much higher precisions are needed. Also, there are regularization algorithms which make C_A well-conditioned in the beginning by adding diagonal corrections to it. In fact, most of the algorithms so far have used an iterative solver to solve , simply because it is rather easy to fit CG into a non-linear inexact Newton algorithm by both gradually reducing the desired precision in the solution and by making the system well-conditioned in the beginning. Therefore I devoted most attention to iterative solvers.

However, I can not overstate the importance of reusing combinatorial data-structures between solutions of . Not only do we need to solve the system repeatedly because of the non-linear iteration wrapping it, but in statistical physics of disordered systems we need to average over many insances of the initial C_A and the supply-demand vector b. So reusing combinatorial data-structures related to the graph G is really of utmost importance, and direct methods are best when it comes to that.

Note to Sivan Toledo concerning TAUCS: *For direct methods, the way to reuse previous solutions of the system for the same graph G would be to reuse the fill-reducing ordering of the nodes and the symbolic Cholesky factorization of C. Before I forget, I must say that although I like TAUCS a lot, the fact that it does not support reusing the symbolic phase of the factorization in its top-level interface bothers me. Since it does support* `taucs_ccs_etree` *though, I am certain it would not take a big change to the library to correct this problem. Dr. Toledo, please let me know if you can change this in TAUCS. If not, I will have to interface to another library or try to understand the internals of TAUCS myself.*

Solving with a direct solver with low desired precision is harder though, because the precision

with which one wants to solve the linear system is not related to the drop tolerance or the drop fill-level in a simple way. In fact, I would appreciate any directions for how to do this. But as I said, reusing combinatorial structures between successive solutions is more imprortant and will compensate for this.

Therefore, the solution time for high precision solutions is not the only indicator of how good a solver is for our applications. The possibility to reuse a previous solution is also important, as well as

the ability to quickly solve a system with low precision. So I will compare and discuss the following aspects of a linear solver/preconditioner:

1. Solving with high precision
2. Solving with low precision
3. Reuse of information computed while solving for previous C_A
4. Memory requirements
5. Ease of implementation and parallelization

Preliminary Timing Results

Before I even start I must give you my results:

In 2D direct complete factorization methods were a clear winner! This is not in fact such a big suprise since these graphs are planar in 2D so nested dissection gives a provably good ordering with logarithmically bounded work and fill-in, since each vertex separator is of length $O(n^{1/2})$. Vaydia's preconditioner also performed rather well, and support tree preconditioners were OK in dealing with the ill-conditioning due to C_A, but still slow.

In 3D preconditioners using support-trees on space-filling curves work well. Complete factorization methods fail badly in 3D because there are no good orderings for the nodes. In 3D even diagonal preconditioning works fine because the distance between the nodes is smaller (i.e. $O(n^{1/3})$) and so AA^T is well-conditioned even for large graphs. I have little experience and knowledge with direct methods.

Question:

For 2D planar graphs, I know that there are very good fill-reducing orderings. For my 3D grid-graphs, I could not get either minimum degree or nested dissection to work well, and the fill was very large. This is probably because of the size of the separator (i.e. $O(n^{2/3})$). *In your knowledge, what do finite-element people do in 3D with direct methods?*

Preconditioners Implemented

Here is a brief description of the preconditioning options and algorithms available in my codes at the moment. They are all support-graph preconditioners to some extent:

Miscaleneous Preconditioners

1. **Diagonal Preconditioner** (`diag_precond`)

   This is of course the simplest support-graph preconditioner, in which the support graph is a single virtual node connected via arcs to all the nodes in the network:

   $$M_{\texttt{diag}} = \mathcal{D}(AC_AA^T) \qquad \#$$

2. **Incomplete Factorization with TAUCS** (`TAUCS_LDLT_precond`)

   For now, temporarily, I also treat complete factorization with TAUCS as a preconditioner for CG (and of course CG converges in 1 iteration). I will change this very soon and make a direct solver a separate option. But TAUCS also offers incomplete factorizations, which I can use as a real preconditioner for CG. The tricky thing to choose then is the drop tolerance and the

fill-reducing ordering method. I do not use modified factorizations here.

Spanning-Tree Based Preconditioners

1. **MST Preconditioner** (`MST_precond`)

   This is a standard preconditioner used in interior-point network optimization codes. It takes as a preconditioning matrix:

   $$M_{\texttt{MST}} = BC_BB^T \qquad \#$$

   where B is the node-arc incidence matrix of the maximal weight spanning tree of G (with conductances used as weights). The support graph here is the MST.

   The cost of this preconditioner comes from finding the MST and then solving with $M_{\texttt{MST}}^{-1}$. I have efficient codes for both, which I coded myself. In fact, I have very efficient codes for rebuilding a maximal spanning tree (forest) using a previously computed almost-maximal spanning tree (forest). Since the weights on the arcs C_A settle down as the non-linear optimization progresses, finding the MST becomes a very fast and efficient step. All the spanning trees and forests which I mention from now on are computed using these reoptimization algorithms!
2. **Rooted MST Preconditioner** (`MST_LDLt_precond`)

   This was also introduced by the network flow community. It basically adds to the MST a rooting virtual node and connects this node to all the nodes in the network. The conductances of the rooting arcs are equal to the sum of the conductances of the non-tree (non-basic) arcs incident on a given node. The resulting preconditioner can be factored very efficiently in linear time and no fill by eliminating the nodes of the tree in tree level order (from leaves up to root),

   $$M_{\texttt{MST\_LDLt}} = BC_BB^T + \mathcal{D}(NC_NN^T) = F_{\texttt{LDLt}}D_{\texttt{LDLt}}F_{\texttt{LDLt}}^T \qquad \#$$

   where N represents the non-tree arcs and FDF^T is a root-free Cholesky factorization of $M_{\texttt{MST\_LDLt}}$.
3. **Incomplete QR MST Preconditioner** (`MST_QR_precond`)

   This is also taken from the network flow community. It is basically a *no-fill* incomplete QR factorization of C using an ordering of the arcs based on the tree level ordering of the MST:

   $$M_{\texttt{MST\_QR}} = F_{\texttt{QR}}D_{\texttt{QR}}F_{\texttt{QR}}^T \qquad \#$$

   I have not found this to be useful to me, and `MST_LDLt` always outperforms it. In fact, it also outperforms `MST_QR`, so that I have really only paid attention to `MST_LDLt`.

Support-Tree Preconditioners

1. **Grebman's Support Tree Preconditioner** (`ST_precond`)
   This is a support-graph preconditioner that is of research interest to most of you. In my codes, the support tree is a regular tree of fixed-predetermined degree k and height h, so that the number of partitions of G needed to construct the support tree is k^h.

Now, assume that we have maximized h for a given k. This means that we want to go all the way to about 1 node per partition. I have interfaced to both CHACO and SCOTCH in my codes, and they are both too slow when one tries to partition all the way up to a single node, which is not suprising, because they were not made for this purpose.
So I decided to look at this from another perspective. Assume we number the leaves of the support trees from 1 to k^h. Then we want to assign (map) $\frac{n}{k^h}$ nodes of G to each of these support nodes. This can be viewed as a partitioning problem, and SCOTCH's features for mapping graphs onto binary tree processor architectures are best suited in terms of providing high-quality support. But these are as I said too costly. Now, assume we have an ordering of the nodes of G that is based on proximity. If we simply chop this permutation into k^h pieces and consecutively assign each section to a partition (support-node), we get a support graph ST. So we can translate the problem of finding a mapping of the nodes to support nodes to a problem of finding a linear embedding of G into 1D space and then taking the nodes in order as they appear in this linear embedding. I hope this is clear.
I have not really achieved much–one still needs a method for computing a good ordering of the nodes. But this makes the code more general and easier to interface with different techniques/libraries. For example, one can partition the graph with CHACO and request a 1D mesh architecture of some size, then order the nodes within each partition using a faster method (for example their default ordering in G), and that way speed up CHACO's partitioning.
But this actually worked quite well when I used a space filling curve to embed G into one-dimensional space, especially if I used a Hilbert curve as shown in the above demo illustrations. For a regular grid, if we set the degree of the support tree to $k = 2^{\dim}$, this produces exactly the support trees Grebman gives in his thesis for regular grids—partition the mesh into $2^{\dim}$ pieces by simply cutting the square (cube) into 4 (8) pieces recursively. For other graphs there may be other ways to do this linear embedding which would work well?
Another very interesting possibility is to use this SFC ordering of the nodes to compute a partition into k^h partitions, and then improve on this partition using standard methods such as KL/FM. To my sad suprise, only CHACO had some support to do this using a global partitioning method called "linear" (which bases the partitions on a median split of the numbering of the nodes in G, which as I said above in my codes is based on a Hilbert SFC).

Note to Pellegrini: *As far as I am aware, SCOTCH is the only serial graph partitioning library which is still in development, and I strongly encourage you to add a graph partitioning method called for example "linear", which will take a permutation (ordering) of the nodes and base the partition on a median split of this ordering. This is useful for two things:*

- *Using a problem-specific fast high-quality initial partitioning:* Although SCOTCH supports a variety of partitioning methods to make an initial guess for KL/FM or multilevel partitioning, in many problems there is a natural physical partitioning which is good and easy to compute, such as in my case space filling curves. By using a node-ordering permutation array provided by the user, SCOTCH can use this other partitioning without knowing anything about it. Is this right?
- *Fast reoptimization of previous partitions:*
  Assume that in the beginning I start with an ordering (linear embedding of the nodes) based on SFC's and use this to create a partitioning for the support tree. This is obviously both connectivity and edge-weight-blind (even though it works wonders). So it would be nice to improve on it without doing too much work (as multilevel would). With the above scheme this can be done. Then in the next iteration, for a new matrix C_A (i.e. new edge weights), I can start with an ordering based on the improved partitioning (as simple as ordering the nodes inside each partition in the same order they were, while ordering the partitions based

upon the mapping produced by SCOTCH with a `mesh1D` or `leaf` architecture). As the weights of the arcs settle down, this step will become faster and effectively reuse previous partitioning information. Does this sound feasible to you?

So, I will test three different method of computing the partitioning of the nodes used in constructing the support tree:

- `ST_SFC`: We use the default Hilbert SFC linear embedding of *G* to map the nodes of *G* onto leaf nodes of the support tree *ST*.
- `ST_CHACO`: I use CHACO with a linear global method and then use a local KL improvement. I also use a 1D mesh processor architecture, which is the closest CHACO has to the tree architecture of SCOTCH. Note that this method requires calling CHACO a new for each new vector of arc conductances C_A, which is excessive. CHACO also has inertial partitioning, which produces similar results to `ST_SFC`, so I will not discuss it futher.
- `ST_SCOTCH`: I use SCOTCH to map the nodes of *G* onto a leaf tree binary processor architecture, using its state-of-the-art multilevel methods. This produces the best partitions and reduces the number of CG iterations the most (I will give numbers later), but it is rather expensive and slow.

Note: I could have also used CHACO or SCOTCH to compute a partitioning of the graph without using edge-weights, and then not changed this partitioning as the conductances of the arcs change (a weight-blind mapping). This should in princinple be better then `ST_SFC`, which is a geometric method that is both connectivity- and weight-blind. I would prefer if SCOTCH some day implements my “linear” method idea, which is in my opinion the best of all worlds.

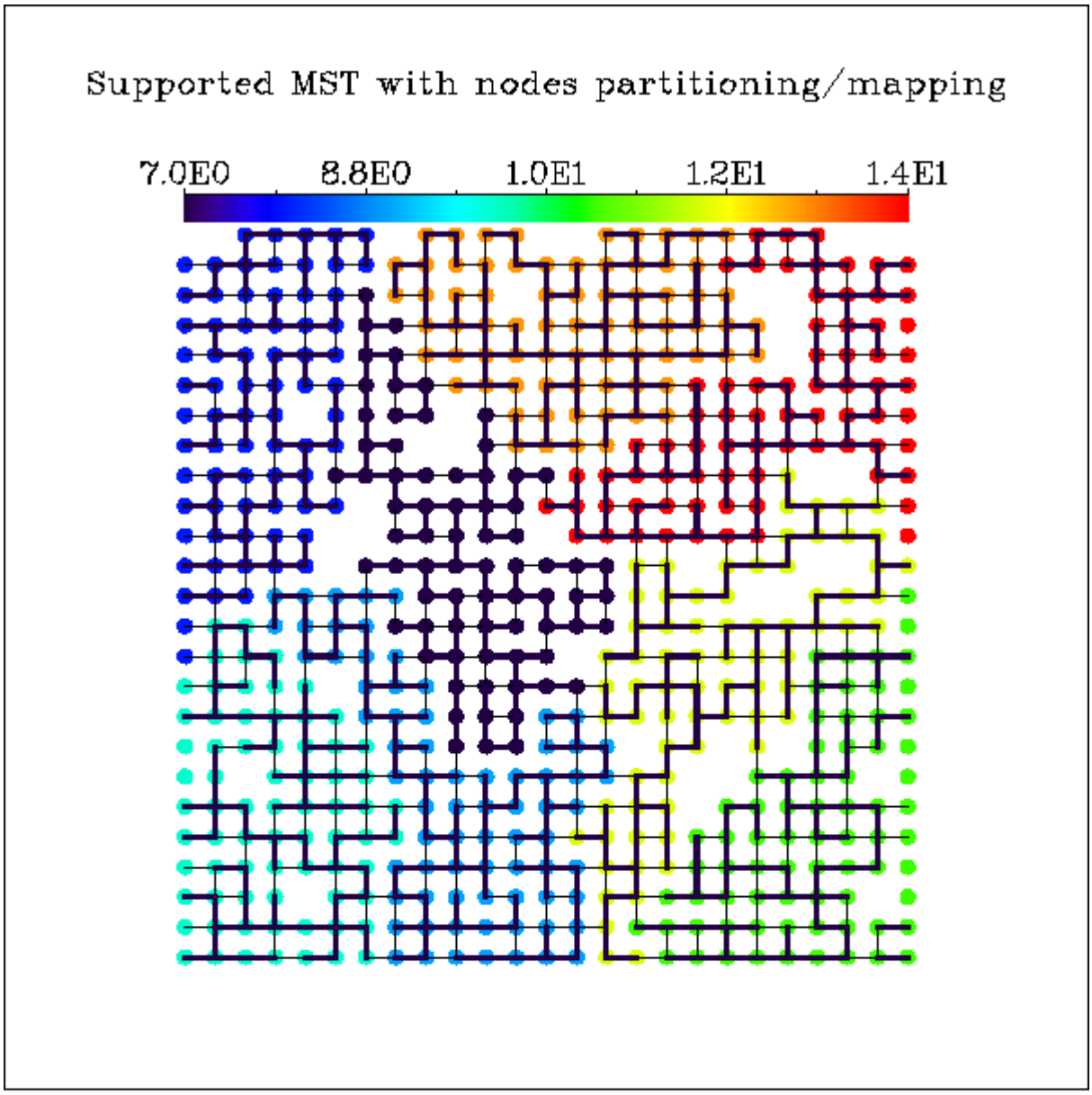
Combined (Transition) Preconditioners

I also implemented/used two preconditioners which act as a bridge between the above subclasses:

1. **Supported MST Preconditioners** (ST_MST_precond)

   These arc as a bridge between `ST_precond` and `MST_LDLt_precond`, and are novel in the sense that I am to my knowledge to first person to try them or implement them, but they are really just a combination of two successful ideas. So let me explain what I did here:

   First, I use SCOTCH to partition the graph into 2^h pieces, using its mapping to a binary tree `leaf` architecture. This produces good and fast partitions as long as 2^h is significantly larger then the number of nodes (say 25-100 nodes per partition). I then use this partitioning to construct a support tree *ST* of degree 2 and height *h* for the graph *G*, just as for `ST_precond` (only now there are a lot more nodes per one leaf support node then before).
   Then, I mask out (remove by a logical mask) all arcs in *G* that cross between subpartitions and then find a minimal spanning forest (MSF) of the resulting subgraph of *G*. This way I build an MST inside each subpartition, but with no arcs crossing between partitions. Finally, I add the arcs of the MSF to the support tree to get a *supported maximal spanning forest* support graph. Here is an illustration which shows the partition of each node with its color and the arcs of the MSF as bold lines. I wish I could plot the support tree in a 3D plot, but this is hard to do (but I hope you can visualize it yourself):


Supported MST with nodes partitioning/mapping
7.0E0
8.8E0
1.0E1
1.2E1
1.4E1

It should be clear that applying (using) this preconditioner is no more difficult or different from first using the `MST_LDLt` preconditioner, then using the `ST` preconditioner. In fact, it was relatively easy to integrate this preconditioner into my codes because I alredy had all the needed code ready.
What are the advantages of this approach over plain support of MST preconditioners? Well, compared to `ST_precond`, `ST_MST_precond` requires partitioning into larger pieces, which is done much faster in all the non-geometric graph partitioning libraries I know of. The above discussions about reusing previous partitioning information applies in full here as well. Compared to `MST_LDLt_precond`, this method seems to reduce the number of CG iterations needed more effectively, simply because a large global MST can not effectively support the whole grid, while smaller spanning trees have more freedom in choosing which arcs to include in the MST (since the global view of the network is not needed any more). Finally, simply the fact that these preconditioners can transient between two radically different support-tree preconditioners, both of which show promise in different situations, is a big bonus. The main defficiency of this approach is that it requires the combined memory overhead of a support tree preconditioner (i.e. overhead of copying my data-structures when interfacing with SCOTCH *and* the spanning tree data structures).

2. **Vaidya's Preconditioner from TAUCS** (`TAUCS_MWB_precond`)
TAUCS is also the first implementation of the maximal weight basis Vaidya preconditioner, and I have tried these as well. These preconditioners can be viewed as support-graph preconditioners which are a transient between `MST_precond` and `TAUCS_LDLt_precond`. They start by constructing an MST of the network, then breaking (partitioning) it into t subtrees of about equal size (in a weight-blind manner—this produces a spanning forest of the graph), and then adding the arc of largest conductance between each pair of trees to the support graph. Finally, a minimum-degree based fill-reducing ordering is applied to the resulting support graph and its conductance matrix is factored completely using TAUCS's (almost) state-of-the-art supernodal/multifrontal Cholesky factorization routines. Please note that I have not at all used the recursive Vaidya preconditioner of TAUCS, since this is too complicated to tune and understand its behaviour.

The main advantage of this preconditioner is that it is a smart, weight-sensitive way of limiting the amount of fill-in. If $t = 1$, there will be no fill since the support graph will be a spanning tree. There are other methods in the literature for limiting fill, but maximal spanning trees have already shown promise in the field of network optimization, where they play a prominent place (since B is a basis for the constraint matrix A). It's main disadvantage is the added overhead of computing a new MST and a new fill-reducing ordering each time a new C_A is used, as well as the fact that there is little *practical* understanding of how t is related to the fill in and the conditioning of the resulting PCG.
I have several suggestions to make along the lines of improving Vaydia's preconditioners:

- Would it help if we also add a virtual rooting node for the spanning forest to the support graph, just like in `MST_LDLt_precond`, and connect it with arcs equal in conductance to the sum of all arcs of G not accounted for by Vaydia's preconditioner. This will cause no major difference from the present implementation since at the end we perform a complete factorization anyway and since this would only add a diagonal correction to the MWB preconditioner. Am I correct in this? This way we get a preconditioner that is a transient between `MST_LDLt` and `TAUCS_LDLt`.
- At present, we must do a full fill-reducing reordering of the nodes at each solution step. This is wasteful, especially for small t. I think Bruce Hendrickson and Sivan Toledo already know this, but let me say it just in case:

Assume we have our spanning forest, and we have already ordered the nodes according to their tree level ordering (i.e. from leafs to roots), and we are about to start adding arcs that go between trees in the forest. Mark all nodes as $\mathcal{F}$. This establishes parent relations for the nodes in the forest, and one node in each tree in the forest is the root (in my codes, I already have these parent relations because they are used when updating and mainting the MST/MSF).When we add such an arc to the support graph, mark the two nodes that it is incident upon and all of their parents in the corresponding tree (i.e. up to the root of the tree) as $\mathcal{T}$. Once we are done adding inter-tree arcs, what we get is a subgraph of G containing the nodes marked as true, $\mathcal{T}$. I hope this can be visualized without a drawing (I can make one if needed)? Let's call this graph $\tilde{G}$. For $t \ll n$, $\tilde{G}$ will be much sparser then G, right? Now notice that only $\tilde{G}$ needs to be reordered again (the $\mathcal{F}$ nodes will all be ordered just before their subtree root, in their tree level order). All other nodes of G can still be ordered based on the spanning forest we started with. Also, the solution during the preconditioning step can be computed on the $\mathcal{F}$ nodes just as for the MST preconditioner, and only on $\tilde{G}$ do we need a complete factorization. Am I correct in this? Further reduction in the size of $\tilde{G}$ is possible by eliminating all nodes in it of degree 2 or less (since it is likely to have long paths inside the spanning trees where the previous forest ordering can also be used).
Wow! This paragraph is too convoluted and long. Anyway, these are just ideas, which are only important when one needs to solve the system repeatedly, like we do. There is another possibly tangible advantage in doing this. If $t = 1$, we know that minimum-degree ordering is the best, since the support tree is a spanning tree. However, for $t = n$, it may be that nested dissection does much better. By doing what I described above, we are free to use any ordering we know of for the nodes of $\tilde{G}$, and always mainting minimal-degree ordering for the nodes that did not get any added arcs. In fact, maybe one can order the nodes of $\tilde{G}$ based on a filled-reduced ordering of the whole graph G, without specifically reordering them again, thus possibly saving a lot of time. What do you think of this?

- Finally, I have a problem with the way the forest is constructed in Vaydia's preconditioners. The first problem is the fact that we start by computing a global MST for the whole graph G. Not only is this expensive, but the global character of the MST constrains the choices for which arc enters the MST a lot (so a heavy arc may have to be thrown away because it forms a cycle even though it is heavy locally, right?). Why not try something like what I did in `ST_MST_precond`: Use a partitioner to partition the graph into t components, and then compute a maximal spanning forest, and finally add inter-tree arcs to make the final support graph. This of course has the disadvantage of needing a partitioner as well, but I suspect it will work better. I have not coded it yet though. Improvements can probably also be made to which inter-tree arcs are chosen to be added to the support graph. At present we choose the heaviest between each pair of trees. This is a simple choice, but I see nothing else special about it.

Preliminary Comparisons

I will give you some preliminary numbers from timing tests I performed. These are indeed very crude (and rude) estimates, since there are many optimizations I know I can do in the codes and

interaces, but which I have not yet done due to a lack of time. I do hope these numbers will give you some better idea of the practical issues involved. All times are in rough seconds, timed on a 350 MHz Pentium II with 512 KB of cache and 512 MB of main memory.

Behaviour of the Condition Number

The conditioning number of depends on three major factors, and here are some quick tests to tell you how. The tests are all performed in 2D with no preconditioning used in CG. A good test for conditioning would observe the error history of conjugate gradient, but this is too time consuming, so below I simply quote the total number of CG iterations needed to converge to a large (say 10^{-6}) precision for the residual. The total number of unpreconditioned CG iterations is very sensitive to the desired precision in this region, though:

Dependence on Conductance Range

In my codes I actually choose the resistances ($R_A = C_A^{-1}$) of the arcs, here from a uniform distribution in the range $[r_{\min}, 1]$. Results for 100 by 100 grid with dilution $d = 0.75$:

$r_{\min}$	# CG iterations
1	510
0.1	750
0.01	1750
0.001	4300

Dependence on Dilution *d*

The lower the dilution (please forgive me for reversing concepts here, usually $1 - d$ is what is called dilution. Just remember that a fraction of $1 - d$ of of the arcs in a complete grid are randomly removed when preparing G), the longer the paths through the network become (but the number of arc and nodes is reduced), so the more CG iterations are needed. Here are some numbers for a 150 by 150 lattice with $r_{\min} = 1$:

d	# CG iterations
1.0	325
0.85	550
0.70	800
0.55	1500

Dependence on Lattice Size *L*

The longer the network, the more nodes, so the more a CG iteration costs. But the conditioning also gets worse because paths through the network become longer. The number of CG iterations is in general proportional to the length of the lattice L. Here are results for $d = 1$, $r_{\min} = 1$:

L	# CG iterations
50	140
100	220
200	360
400	560

Support-Tree Preconditioners

Now that we know how the conditioning depends on the physical parameters, let me say how different preconditioners cope with these effects. First I focus on support tree preconditioners. Quick tests indicated that in 2D the degree of the support tree should be 4, which is expected ($2^{\dim}$ in general), and the height in these experiments is chosen so that we have as close to 1 node per partition as possible. For SCOTCH with `leaf` (tree) architecture the degree must be 2.

The first thing to test is the influence of the partitioner (linear embedding really, as I discussed at length above). Here are some different partitioners for a 200 by 200 grid with $d = 0.75$, $r_{\min} = 0.001$ (so a variation of 1000 in the conductances). I record the time it takes to compute the partitioning (combinatorial reoptimization), the time spent on applying the preconditioner, the total time spent in PCG, and the number of iterations in PCG:

Partitioner	Embedding	Preconditioning	Total PCG	# iterations
CHACO inertial+KL	2.6	0.33	5.0	190
CHACO linear+KL	4.5	0.35	5.6	160
CHACO intertial only	0.3	0.70	2.4	330
SCOTCH `gfx`, `leaf`	2.9	0.50	3.9	120
SCOTCH multilevel	2.1	0.85 (???)	3.9	200
SFC ordering (degree 2)	0.0	0.90	2.0	240
SFC ordering (degree 4)	0.0	0.60	1.7	260

Now, I have not tuned all the options in the partitioners. But it is clear that partitioners take a long time compared to the actual CG iteration. SFC-based ordering seems the best/ overall. However, do notice that a good partitioning does significantly reduce the number of PCG iterations (SCOTCH's multilevel is presently tuned for `ST_MST_precond`, that is why it seems not to work as well above. Otherwise it is the best in terms of quality). For comparison, diagonally preconditioned PCG takes 730 iterations and a total of 3.5 seconds, 0.35 of which are spent preconditioning.

Supported MST Preconditioner

Partitioning Method

For these I eventually decided to use SCOTCH with multilevel mapping to a tree processor achitecture, and to use a support tree of degree 2 (this is a must with this option in SCOTCH). Here is a quick table to show you why, giving the partiotioner, the time to compute the partitioning, the total time to solve the linear system and the number of PCG iterations for a 200 by 200 lattice with

$d = 0.75$, $r_{\min} = 0.001$, and a support tree of height 5 (meaning $2^5 = 32$ partitions, or about 100 nodes per partition):

Partitioner	Partitioning time	Total PCG	# iterations
CHACO (inertial+KL)	0.42	3.0	240
SCOTCH (multilevel)	0.40	1.9	140
Hilbert SFC	0.0	2.2	190

This shows you that the quality of the partitioning is not of utmost importance, but it does help convergence of course.

Size of Partitions

The next important thing to test is the number of partitions, i.e. the number of nodes per partition (or approximately per subtree in the spanning forest). Here is this for the previous lattice. We change the number of partitions by changing the height of the support tree h (as 2^h):

h	Partitioning	Preconditioner	Total PCG	# iterations
1	0.11	1.30	2.3	180
5	0.40	1.40	1.8	130
7	0.47	1.10	1.6	100
9	0.55	1.60	2.2	140

Although this parameter seems somewhat important, it does not seem critical, and in fact I observed that about 50-150 nodes per partition was optimal in most cases (even in 3D), although I did not experiment too much with this.

Here are some the other MST-based preconditioners for comparison:

Preconditioner	Total PCG	# iterations
`MST`	2.5	240
`MST_QR`	4.2	400
`MST_LDLt`	1.7	160

And here is a comparison of three preconditioners for a 500 by 500 lattice (the rest the same as above):

Preconditioner	Total PCG	# iterations
`MST_LDLt`	208	480
`ST` with SFC embedding	125	500
`ST_MST` with $h = 11$	90	190

My conclusion would be that if the partitioner is sped up a bit by reusing partitioning information from previous iterations then this is better than support trees and probably more robust. Note that these also do not require too much memory. Plain support trees require $O(n \log n)$ storage to store the whole support tree. Here the storage is almost linear (but the spanning tree requires quite a bit). Also, remember that it is easier to compute the partitioning for larger partitions, in particular, SCOTCH works just fine in this regime! I believe this effectively improves upon the previous work of Grebman, at least for more irregular and ill-conditioned matrices such as ours. Also, it is an improvement over the work in the interior point network flow community that I have discussed with Bruce Hendrickson. What do you think?

TAUCS Preconditioners

Here are some timing tests for complete Cholesky factorizations and Vaidya's preconditioners. The numbers are very pleasing for these direct methods, at least in 2D. Later I will compare all of the preconditioners against one another and look at 3D.

Complete LL^T Factorization

The most important thing here is of course the ordering method. Here is a table comparing methods, giving the time to order, factorize, apply (solve), and the fill factor (number of non-zero elements in the factor L) for a 250 by 250 lattice ($d = 0.75$, $r_{\min} = 0.001$ as usual). For comparison, the number of non-zero elements in the original conductance matrix is 1.5×10^5:

Ordering/preconditioner	Ordering	Factorization	Solve	Fill (in 10^5)
`genmmd`	2.2	2.4	0.8	5.4
`md, amd, mmd`	too long			
`metis`	4.6	2.42	0.9	5.7
`identity`	0.0	broke down		too large
`ST_precond` with SFC	0.0		17.0	370 iterations
`ST_MST_precond`, $h = 9$	1.3 (init.)	4.0 (partitioning)	16.0	140 iterations

Based on my experimentation, `genmmd` is a very good ordering routine. METIS is not bad either, but SCOTCH really has a lot more options and functionality. I have not yet used SCOTCH to order TAUCS matrices though, because this requires some double interfacing. But because the results above are so encouraging, I will devote a separate routine to direct factorizations and do this, especially when/if TAUCS provides routines for reusing the symbolic phase of the factorization. For the above example, complete factorization takes about 2.2+2.4+0.8=5.5 seconds, as compared to 15 or 16 for support tree preconditioners. When one takes into account that the reordering phase can be reused as well as part of the factorization, this ratio becomes an order of magnitude difference between direct and iterative methods!

Here is a table for a 500 by 500 lattice (about 6×10^5 zeros in C):

Ordering/preconditioner	Ordering	Factorization	Solve	Fill (in 10^5)
`genmmd`, supernodal	9.6	11.5	3.0	25
`metis`, supernodal	21.6	11.6	3.0	26
`genmmd`, multifrontal	9.6	18.6	2.5	25

Again, even for this huge lattice, multiple minimum degree is better. Also notice that supernodal routines offer little speed up over multifrontal. I have also observed that these offer little speed up over plain Cholesky factorization. In my reading, I have read that if the matrix and the factors are very sparse, dense-kernel-based factorizations offer little speed up. This is the case I think for our lattices in 2D at least.

Incomplete factorizations in 2D worked on my machine up to about 1000 by 1000 lattices (order million nodes), and quite fast indeed. For these sizes, ordinary PCG takes way too long (I was impatient to wait for it), no matter how you precondition. For such a large lattice with $d = 0.75$, ordering took about 50 seconds, factorization took 60, solution 8.5, to give a total of about 120 seconds. To us this is impressive compared to PCG which physicists most often use. In terms of

memory, the total memory that my Fortran program allocates and uses for everything is about 150 MB, while the maximum memory used during the solution was 400 MB or so, almost two thirds of which is the factorization. But it is worth it when one looks at the speed obtained, and memory is getting cheaper every day.

TAUCS Point-Incomplete Cholesky

Here the thing to test is the influence of the drop factor. Here is a short demo table for our 250 by 250 network from above (the droptolerance is obviously related to the range of arc conductances):

droptol	# iterations	Ordering	Total PCG	Fill (in 10^5)
10^{-6}	1	2.2	2.6	5.3
10^{-4}	6	2.2	2.8	4.7
10^{-3}	25	2.2	4.4	3.9
10^{-2}	80	2.2	10.6	2.9

It does not seem to effective to me. And since we can not reuse the symbolic factorization here, I vote against incomplete factorizations of this type. There are better methods in the literature though. I think though that Vaidya's preconditioners are more promissing in terms of controlling fill-in, so I proceed to give some numbers for them:

TAUCS Vaidya's Preconditioner

As I said earlier, this preconditioner is not impemented optimally yet in my opinion, with room for improvement. But some testing results of mine will hopefully benefit your research. The main thing in these preconditioners is of course the number of subtrees t. In my codes, I specify the subtree ratio $\tau = t/n$ (so $\tau = 0.01$ implies about 100 nodes per subtree). Here are some numbers for our 250 by 250 lattice, with a total of about 150,000 non-zeros in the conductance matrix. Some of these timings are dirty, i.e. they include in them things other than what is written:

τ	Actual t	# iterations	Ordering	Factorization	Total PCG	Fill (10^5)
1.0	140,000	10	1.3	5.8	8.7	4.7
0.1	120,000	25	1.3	5.0	10.7	2.6
0.05	2400	34	1.3	4.8	12.0	2.1
0.01	500	67	1.2	4.5	17.5	1.6
0.0	1	630	1.1		120	1.4
`MST_precond`	1	660		3.7	65	

These number clearly show that as far as speed in 2D is concerned, the more memory used–the faster. However, Vaidya's preconditioners with about 20-50 nodes per tree have in my experiments shown to markedly reduce fill-in while also significantly reducing the number of CG iterations needed. So in 2D these are quite promissing!

In fact, in 2D these work much better than my other no-fill support graph preconditioners. For example, here is a 500 by 500 lattice near percolation ($d = 0.501$). This is very difficult for CG codes due to the fractal nature of paths between nodes:

Preconditioner	# iterations	Preconditioning	Total PCG
`ST` with SFC	4000	62	190
`ST_MST` ($h = 9$)	2600	140	85
`TAUCS_MWB` ($\tau = 0.05$)	45	50	10

Here is the same table not near percolation ($d = 0.75$):

Preconditioner	# iterations	Preconditioning	Total PCG
`ST` with SFC	510	42	140
`ST_MST` ($h = 10$)	320	88	170
`TAUCS_MWB` ($\tau = 0.05$)	47	34	67

These conform to general expectations: Support trees based on SFC's do not work well near percolation, while supported MSF's work better. However, Vaidya's preconditioner outperforms (do not forget that complete factorization in this case outperforms all anyway).

Performance Inside a Non-Linear Optimization Solver

As I said in the beginning, we use this linear solver inside a non-linear Newton-based network optimization code. Here is what a typical output looks like for a cost function that is of the form $f_i(x) = \alpha_i x^3$, where the α_i are sort of resistances of the arcs (the resistances here actually change as the optimization proceeds) in the interval $[0.1, 10.0]$ and distributed uniformly.

In 2D

Here are some logs from my optimization library for the above cost function for a 250 by 250 lattice with $d = 0.75$.

When using **complete factorization** throughout:

```
[hpf@gauss 2Ddp]$ TestSSCNO.x

_____________________________________Start of TCGN statistics
    Iter.#         Excess   LS step  LS #iter    CG residual  CG #iter
_____________________________________________________________________

         1        1.00000     2.391         4      0.915E-10         1
         2        1.39781     0.807         5      0.182E-10         1
         3        0.68251     0.772         6      0.227E-10         1
         4        0.33752     0.670         6      0.177E-10         1
         5        0.13498     0.702         6      0.505E-11         1
         6        0.06248     0.657         9      0.357E-11         1
         7        0.02515     0.690         6      0.115E-11         1
         8        0.01154     0.642         9      0.104E-11         1
         9        0.00478     0.685        10      0.644E-12         1
        10        0.00237     0.635         6      0.689E-12         1

_____________________________________End of TCGN statistics

 Profiling timing results:
 _____________________________________________:
 SSCNO initialization                         : 0.890000105
```

```
---> Preconditioner initialization            : 2.15000010
TCGN iterations                               :
--->Array updates                             : 0.709990501
--->Conductance calculation                   : 2.96000242
--->Line Search                               : 46.1000023
------>Cost function evaluation              : 44.4799957
------>Root bracketing                        : 5.82000589
------>Root finding solver                    : 40.1699905
--->Solving Newton's linear system            : 30.5200043
------>Preconditioner construction           :
--------->Combinatorial reoptimization        : 0.00000000E+00
--------->Creation (factorization)            : 24.1600018
------>PCG iteration                          : 5.36999989
--------->Preconditioning                     : 4.53001308
--------->Dot products                        : 0.109998226
--------->Vector updates                      : 0.269987583
--------->Matrix-vector products              : 0.390001297
_____________________________________________:
Total elapsed-time timings for  89963 arcs and  58641  nodes
_____________________________
Creating the network problem took    : 1.63000000
Solving the network problem took     : 81.2300034
Cost function evaluations took       : 47.4399986

_____________________________
Average arc conductance at solution: 0.2796630287808338
Extreme conductances at solution: 5.301579178946971E-03 199.7359795070896
At present allocated:  0  bytes, maximum allocated at one time:  13768831  bytes,
```

Some of these numbers are not of interest to any of you since they are related to the non-linear network solver. For us we want to focus on the section about the linear solver in the inner iteration:

```
--->Solving Newton's linear system            : 30.5200043
------>Preconditioner construction           :
--------->Combinatorial reoptimization        : 0.00000000E+00
--------->Creation (factorization)            : 24.1600018
------>PCG iteration                          : 5.36999989
--------->Preconditioning                     : 4.53001308
--------->Dot products                        : 0.109998226
--------->Vector updates                      : 0.269987583
--------->Matrix-vector products              : 0.390001297
```

As I said earlier, if PCG is used here, then it is possible to use a lower precision in the beginning of the iterations. This effectively reduces the number of CG iterations needed. Also, as the optimization proceeds toward the optimum, the initial guess in PCG becomes closer to the true solution. Complete factorization could not use any of these facts explicitly.

Here is what we get when using **support tree preconditioning** with SFC based partitioning:

```
--->Solving Newton's linear system            : 46.8600044
------>Preconditioner construction           :
--------->Combinatorial reoptimization        : 0.00000000E+00
--------->Creation (factorization)            : 0.890007734
------>PCG iteration                          : 45.8900108
--------->Preconditioning                     : 13.3700991
--------->Dot products                        : 5.80995893
--------->Vector updates                      : 10.4799614
--------->Matrix-vector products              : 13.7399788
```

So the difference is not that great any more. Similar results are observed for **supported MSF preconditioning** with $h = 9$:

```
--->Solving Newton's linear system            : 86.1699982
------>Preconditioner construction           :
--------->Combinatorial reoptimization        : 45.2400436
--------->Creation (factorization)            : 0.769967079
------>PCG iteration                          : 40.0799789
--------->Preconditioning                     : 23.4500427
```

```
 --------->Dot products                        : 2.89997721
 --------->Vector updates                      : 5.26985025
 --------->Matrix-vector products             : 7.08008862
```

And for **Vaidya's preconditioner** with $\tau = 0.05$:

```
--->Solving Newton's linear system            : 74.1699829
 ------>Preconditioner construction           :
 --------->Combinatorial reoptimization       : 0.00000000E+00
 --------->Creation (factorization)           : 55.3799896
 ------>PCG iteration                         : 17.5900021
 --------->Preconditioning                    : 13.4600077
 --------->Dot products                       : 0.829998016
 --------->Vector updates                     : 1.27001905
 --------->Matrix-vector products            : 1.70000124
```

In 3D

The picture is much different in 3D. There iterative solvers gain advantage because the distance between nodes is of order $O(n^{1/3})$ instead of $O(n^{1/2})$ as in 2D. However, direct factorizations gain a big disadvantage, because separators become $O(n^{2/3})$ instead of $O(n^{1/2})$ as in 2D.

Here is the same cost function as above with a 25 by 25 by 25 lattice with $d = 0.80$ with periodic boundary conditions in the *y* and *z* directions (this gives a further disadvantage to factorizations):

Complete factorization as a preconditioner:

```
 --->Solving Newton's linear system           : 154.230011
 ------>Preconditioner construction           :
 --------->Combinatorial reoptimization       : 0.00000000E+00
 --------->Creation (factorization)           : 149.430023
 ------>PCG iteration                         : 4.60998154
 --------->Preconditioning                    : 4.39996910
 --------->Dot products                       : 3.99932861E-02
 --------->Vector updates                     : 7.00206757E-02
 --------->Matrix-vector products            : 7.00092316E-02
```

Suprisingly slow! A look at the log from TAUCS at the last iteration shows why—the fill is just very large in this case:

```
taucs_ccs_genmmd: calling genmmd, matrix is 15624x15624, nnz=53034
taucs_ccs_genmmd: genmmd returned.
                Symbolic Analysis of LL^T: 2.92e+06 nonzeros, 1.12e+09 flops
                Symbolic Analysis            =      0.176 seconds (0.180 cpu)
                Supernodal Multifrontal LL^T =     21.510 seconds (21.390 cpu)
```

Nested dissection does not do much of a better job reordering either, as seen when trying to use METIS for the fill-reducing ordering:

```
taucs_ccs_metis: calling metis matrix is 15624x15624, nnz=53099
taucs_ccs_metis: metis returned
                Symbolic Analysis of LL^T: 1.90e+06 nonzeros, 4.52e+08 flops
                Symbolic Analysis            =      0.147 seconds (0.150 cpu)
                Supernodal Multifrontal LL^T =     17.016 seconds (16.880 cpu)
```

Here is **Vaydia's preconditioner** with $\tau = 0.01$ ($\tau = 0.05$ was too slow in this case):

```
 --->Solving Newton's linear system           : 18.3700027
 ------>Preconditioner construction           :
 --------->Combinatorial reoptimization       : 0.00000000E+00
 --------->Creation (factorization)           : 12.8500004
 ------>PCG iteration                         : 5.29999828
 --------->Preconditioning                    : 4.10999680
 --------->Dot products                       : 0.219999790
 --------->Vector updates                     : 0.189996719
 --------->Matrix-vector products            : 0.720005035
```

A look at the log from TAUCS at the last iteration:

```
taucs_ccs_genmmd: starting (genmmd)
taucs_ccs_genmmd: calling genmmd, matrix is 15626x15626, nnz=32340
taucs_ccs_genmmd: genmmd returned.
               Symbolic Analysis of LL^T: 7.02e+04 nonzeros, 8.54e+05 flops
               Symbolic Analysis            =      0.048 seconds (0.050 cpu)
               Supernodal Multifrontal LL^T =      0.431 seconds (0.430 cpu)
```

So we see that the success of Vaidya's preconditioner is tied to the success of complete factorizations—if there are good elimination orderings, then it will perform well, if not, it won't! So I prefer to look at this preconditioner as a way to control the fill-in for factorizations in 2D mostly. What do you think?

Now the suprise. Ordinary PCG performs very well on this problem. The only preconditioner which works well are support trees of degree 8 and plain diagonal preconditioning:

SFC-based **support tree preconditioner** (tree degree 8):

```
--->Solving Newton's linear system            : 4.33000040
------>Preconditioner construction            :
--------->Combinatorial reoptimization        : 0.00000000E+00
--------->Creation (factorization)            : 0.289999723
------>PCG iteration                          : 4.04000044
--------->Preconditioning                     : 0.919996500
--------->Dot products                        : 0.370002389
--------->Vector updates                      : 0.760006666
--------->Matrix-vector products              : 1.72999382
```

Supported MSF preconditioner with $h = 7$ (not good either?!?):

```
--->Solving Newton's linear system            : 15.9999981
------>Preconditioner construction            :
--------->Combinatorial reoptimization        : 12.8199921
--------->Creation (factorization)            : 0.300005198
------>PCG iteration                          : 2.87000012
--------->Preconditioning                     : 1.64999652
--------->Dot products                        : 0.189999819
--------->Vector updates                      : 0.189998865
--------->Matrix-vector products              : 0.750004053
```

In fact, simple **diagonal preconditioning** does sufficiently well in this case:

```
--->Solving Newton's linear system            : 4.89999771
------>Preconditioner construction            :
--------->Combinatorial reoptimization        : 0.00000000E+00
--------->Creation (factorization)            : 9.00001526E-02
------>PCG iteration                          : 4.78999710
--------->Preconditioning                     : 0.529995203
--------->Dot products                        : 0.490005016
--------->Vector updates                      : 1.02999663
--------->Matrix-vector products              : 2.43999815
```

I get similar results for larger systems as well. I will swamp you a little bit with too much data here, but here are logs from diagonal and ST preconditioning for a 50 by 50 by 50 equivalent of the previous lattice:

Diagonal:

```
--->Solving Newton's linear system            : 54.4099998
------>Preconditioner construction            :
--------->Combinatorial reoptimization        : 0.00000000E+00
--------->Creation (factorization)            : 0.669992924
------>PCG iteration                          : 53.6000214
--------->Preconditioning                     : 5.61999464
--------->Dot products                        : 7.43007994
--------->Vector updates                      : 13.7700024
--------->Matrix-vector products              : 23.9698772
```

Degree-8 **SFC support tree**:

```
--->Solving Newton's linear system            : 43.8099976
------>Preconditioner construction            :
--------->Combinatorial reoptimization        : 0.00000000E+00
```

```
--------->Creation (factorization)           : 2.44000292
------>PCG iteration                         : 41.2599983
--------->Preconditioning                    : 13.2199602
--------->Dot products                       : 4.31003094
--------->Vector updates                     : 7.81012678
--------->Matrix-vector products            : 14.0800467
```

One might say that diagonal preconditioning is enough. But this is not really the case if C_A is significantly more ill-conditioned at the solution point. For example, here are the same two logs from above, but now with initial range for the "resistances" α_i $[0.01, 100.0]$ (before it was $[0.1, 10.0]$):

Diagonal:

```
--->Solving Newton's linear system          : 118.730057
------>Preconditioner construction          :
--------->Combinatorial reoptimization       : 0.00000000E+00
--------->Creation (factorization)           : 1.06001854
------>PCG iteration                         : 117.420013
--------->Preconditioning                    : 11.4601526
--------->Dot products                       : 15.2589092
--------->Vector updates                     : 26.7803822
--------->Matrix-vector products            : 57.5801201
```

And for **support-trees**:

```
--->Solving Newton's linear system          : 91.5899506
------>Preconditioner construction          :
--------->Combinatorial reoptimization       : 0.00000000E+00
--------->Creation (factorization)           : 3.09995985
------>PCG iteration                         : 88.3000336
--------->Preconditioning                    : 19.9103394
--------->Dot products                       : 9.64003468
--------->Vector updates                     : 17.5898857
--------->Matrix-vector products            : 37.0296402
```

So support trees are at least more stable in this respect, and I recommend them for 3D problems. What do you make of all the above data I have bothered you with?

Summary and Future Directions

This is an exiting and novel field. I wish I had more time to work on it. But I must continue with coding my library now. I hope to do more work on preconditioning and linear solvers in general in the future though, and please keep me apraised on any progress you learn of or do yourselves. Here are some of my conclusions:

It seems that three kinds of codes are necessary in any really state-of-the-art network optimization software:

- **Graph Partitioners**: As far as serial codes go, I vote for SCOTCH. I hope this library is further updated, developed, advertised and distributed, and in particular I hope that the linear partitioning method is added to it. It's ability to map to a tree processor architecture is also useful in the context of support trees. It's interface is very nice and efficient, and as an added bonus the new version has state-of-the-art fill-reducing ordering routines that use either nested dissection or minimum degree algorithms, and some novel hybrid schemes. Combined with ParMETIS, this system can do wonders on a distributed-memory parallel machine.
- **Maximal Spanning Forests** codes: I have state-of-the-art codes for this that can reoptimize previously computed spanning trees and do other useful things. i do not plan to release them separately as a library, but anyone is welcome to take a look at them. MST's are important to sparse network optimization mostly becase they are a robust way to involve the arc weights in deciding which arcs to drop from an incomplete factorization, as in Vaydia's preconditioner. They are also very well studied.

- **Sparse Cholesky Factorization** codes: I know little about this, but TAUCS is a nice library and it implements the two major contestants, supernodal and multifrontal techiques. It is of course not at the stage that other libraries are yet. The wonderful thing is that there are many public-domain codes in this field, thanks to finite-element applications. Adding functionality for reusing the symbolic factorization phase in TAUCS is very important!

Finally, let me summarize and compare the three major preconditioner contestants and direct factorization along the points I said I would focus on:

1. **Solving with high precision:**

   In 2D, direct solvers are a clear winner. Vaydia's preconditioners work OK, but for us direct factorization is better because one can very effectively reuse information from a previously computed solution. I made suggestions on how to improve Vaydia's preconditioners. Support trees are effective in controlling the ill-conditioning due to ill-conditioning in C_A, but are still slow. The main reason for this is that the number of iterations needed in unpreconditioned CG is of the order $O(n^{1/2})$, so for the largest networks we can do (which is what interests us most), too many CG iterations are needed, regardless of the no-fill preconditioner used.
   In 3D, no preconditioner seems to justify its cost fully. Support trees based on space-filling curves are at least easy to construct and reuse, and are effective in controlling the ill-conditioning due to ill-conditioning in C_A, so I at least recommend using them. I am not sure what else to say?
2. Solving with low precision

   PCG is ideal for this because it directly computes the residual. But it is just too slow in 2D, and offers little chances of reusing previous solutions. I can offer no good ways of controlling the ammount of work needed in constructing any of the successful preconditioners based on the desired precision. We expect for example that a small t should be used for lower precisions, and larger for larger precisions in Vaidya's preconditioners. But how exactly? I can only say that for our specific needs the reuse of combinatorial information is more important and thus I will throw this into the water and only save time by reducing the number of PCG iterations needed, not so much the cost of constructing the preconditioner.
3. Reuse of information computed while solving for previous C_A.
   Direct methods are clearly best in this. But in 3D they seem infeasible due to large fill in? There are three main forms of reusing previous information for the other preconditioners
   - *Reusing previous partitioning information*, probably by performing incremental KL/FM reoptimization. I have found KL/FM to be somewhat unstable in both CHACO and SCOTCH—it hangs every now and then. Some of the tricks you use there, such as using arc weights as indices and logarithmic indexing in SCOTCH, are somewhat dangerous and easy to not take into account properly. My suggestion is the linear global method idea. `ST_precond`, `ST_MST_precond` and `TAUCS_MWB_precond` can all benefit from this.
   - *Reusing the previous MSF*—I have codes to do this based on algorithms developed for the network simplex method. I have found them just as fast or faster then codes that start from scratch even when one reoptimizes a completely random spanning forest.
   - *Reusing fill-reducing orderings* information with Vaydia's preconditioners. I gave some suggestions above on how to do this, but I am really not certain of it.
4. **Memory requirements**
   Direct methods are by common wisdom the worst here. But my observations are that in 2D they

require about the same storage as using a preconditioner which requires additional data structures, such as `ST_precond` or `MST_ST_precond`. In 3D the fill was not acceptable. I like the idea's behind Vaidya's preconditioners a lot–they combine graph partitioning, MSF's, ordering and factorizations all in one, and provide a descent guard against excessive fill. But they did not seem to yet be worthwhile in 3D, especially considering how much work is involved in implementing them. But there is *plenty* of room for improvement here!

5. **Ease of implementation and parallelization.**

   Iterative methods are of course a winner here. All support graph preconditioners are to some extent paralleliziable, since one can always exclude inter-processor arcs from the support graph, or use a support tree which is truncated after each processor holds about one support node or so. There is a lot of work done by you and others in this field. Unfortunately, I will not have time to parallelize the codes for now. One of the main obstacles for me is the lack of a good programming paradigm–OpenMP is too restricted, and I truly dislike MPI, and my personal love, HPF, is in dying stages... But there is definitely promise in this respect in all of the preconditioners discussed. It seems though that for distributed machines one should always keep an iterative solver as the final wrapper around anything that one does serially on each processor. Bruce Hendrickson would be a person to decide this.

In fact, the overall conclusion is the same as many others before me have come to: The best solution technique and preconditioner depends on the exact specifics of the problem at hand. But a combination of powerful general-purpose tools from graph algorithms does seem a promising direction to look into.

Thank you all for your help so far. I hope we can further improve and develop upon what has been done so far.